Relatório Projeto Final

Maria Eduarda M. de Holanda - 19/0043725

24 de Outubro de 2021

1 Introdução

Programação Concorrente é um paradigma de programação no qual as instruções computacionais ocorrem de forma simultânea, isto é, no mesmo tempo lógico de execução. Um programa concorrente especifica dois ou mais processos que cooperam na realização de uma tarefa. Cada processo é um programa sequencial que executa uma sequência de instruções [1]. Eles dão suporte à possibilidade de haver operações concorrentes mesmo quando há apenas uma CPU disponível, transformando uma única CPU em múltiplas CPUs virtuais [3].

A programação concorrente é mais complexa que a programação sequencial, pois, além de existir os erros comuns dos programas sequenciais, há ainda erros associados às interações entre processos. O desafio na programação concorrente vem da necessidade de sincronizar a execução de diferentes processos e permitir que eles se comuniquem [2]. Apesar disso, existem muitas áreas nas quais a programação concorrente é vantajosa, uma vez que é possível aumentar a quantidade de tarefas executadas em um determinado período de tempo, aumentando o desempenho dos programas.

Quando falamos em Programação Concorrente, é importante ter em mente que processos, quando estão trabalhando juntos, podem compartilhar de alguma memória comum. Durante o semestre, foram estudadas diversas maneiras de gerenciar o uso de memória compartilhada, de forma evitar que condições de disputa, isto é, impedir que dois ou mais processos acessem um mesmo recurso ao mesmo tempo, estabelecendo uma exclusão mútua entre eles. O objetivo desse trabalho é idealizar um problema do dia a dia que envolve disputa entre processos e desenvolver um algoritmo para tratar esses diferentes tipos problemas de comunicação que envolvem uma memória compartilhada.

2 Formalização do Problema Proposto

O problema proposto consiste numa simples implementação de um canil, para adoção de cachorros. Foram criados três processos que comunicam entre si, as carrocinhas, os clientes e um funcionário. As carrocinhas ficam a todo momento na rua procurando cães perdidos para levá-los para o canil. Ademais, a cada instante, chegam novos clientes ao canil com o desejo de adotar novos cãezinhos. Os clientes, como possuem muita vontade de adotar, mesmo se o canil estiver vazio, em vez de irem embora, eles preferem aguardar na sala de espera o momento que um novo cachorro chegar.

Existe somente um funcionário que irá atender os clientes. O funcionário fica esperando novos clientes chegarem ao canil e quando tiver algum cachorro para ser adotado ele irá atender o cliente que está na sala de espera. A sala de espera do canil possui uma lotação máxima, ou seja, quando a capacidade de clientes na sala de espera estiver cheia, novos clientes que chegarem ao canil deverão ir embora, pois não terão onde ficar. Dessa forma, sempre que o funcionário atender o cliente, esse cliente liberará um espaço na sala de espera para que outro futuro cliente possa esperar sua vez de adotar.

3 Descrição do Algoritmo Desenvolvido

Para desenvolver o algoritmo para a solução do problema proposto, foram utilizados *locks*, variáveis de condição e semáforos. Antes de iniciar o problema de fato, é necessário na função principal criar as *threads* que serão utilizadas no programa. Dessa forma, é criada uma *pthread* para um funcionário, e dois vetores de *pthreads* para clientes e carrocinhas, ambos com tamanhos previamente definidos. Além disso, é necessário também inicializar os semáforos que serão utilizados no programa. O semáforo da sala de espera é inicilizado com o tamanho máximo de clientes permitidos na sala, uma vez que ela começa vazia. Já os semáforos que sinalizam se o funcionário está atendendo um cliente e se o cliente já foi atendido começam ambos com 0, pois nenhuma dessas ações está acontecendo a priori.

As *threads* das carrocinhas e dos clientes compartilham uma variável global `canil`. Para evitar uma condição de corrida, o acesso à essa variável é feito através de um *lock*. Além disso, foram utilizadas duas variáveis condicionais para sinalizar o acesso e liberação dessas duas *threads*, uma vez que se o `canil` estiver cheio, as carrocinhas deverão parar de trabalhar e se o `canil` estiver vazio, os clientes devem esperar. Ainda,

quando as carrocinhas chegam ao `canil` após encontrar um cachorro e antes ele estava vazio, elas chamam todos os clientes que estavam esperando na sala de espera. Os clientes também sempre avisam as carrocinhas após a adoção, para que as carrocinhas nunca parem de procurar mais cãezinhos.

Além disso, existe também a *thread* do funcionário, que se relaciona com as *threads* dos clientes. Foram utilizados três semáforos para controlar o fluxo de clientes que são atendidos pelo funcionário e também o fluxo de clientes que estão esperando a sua vez de serem atendidos. O conceito de semáforo é motivado por uma das maneiras pelas quais o tráfego rodoviários é sincronizado para evitar acidentes de trânsito. Os semáforos concorrentes, da forma parecida, podem ser vistos como mecanismos que sinalizam as condições a fim de garantir a ocupação mutuamente exclusiva de regiões críticas [1].

O primeiro semáforo utilizado determina a quantidade de clientes que estão aguardando na sala de espera, e uma vez a sala de espera fica cheia, se novos clientes chegarem ao canil eles devem ir embora, já que não há espaço para eles aguardarem. Existem outros dois semáforos que determinam a relação dos clientes com o funcionário. Sempre que é a vez do cliente ser atendido, ele libera primeiro a vaga que ele estava ocupando na sala de espera para um próximo cliente e então vai ser atendido pelo funcionário. Primeiro, o cliente precisa acordar o funcionário para atendê-lo e então ele espera até ser atendido. Uma vez que o funcionário acabar de atendê-lo, ele então adota um cãozinho e vai embora. Por isso, são utilizados dois semáforos, um que sinaliza que o funcionário está atendendo, e outro que sinaliza que o cliente foi atendido.

É válido ressaltar também, que nem todas as ações feitas pelas carrocinhas ou pelos clientes são regiões críticas, ou seja, fazem parte do código do programa onde é feito o acesso à memória compartilhada. Cada carrocinha começa procurando um cachorro fora do canil e, ao achar um cachorro perdido, elas retornam ao canil. Somente quando a carrocinha retorna ao canil ela entra região crítica para adicionar mais um cão na variável compartilhada `canil`. A mesma situação se aplica ao cliente, o cliente antes de chegar ao canil para adotar não se encontra na região crítica do código, somente quando ele consegue entrar na sala de espera do canil que isso acontece.

4 Conclusão

Durante o semestre, foram vistas diversas formas de trabalhar e otimizar processos concorrentes. O projeto proposto teve como objetivo prin-

cipal colocar em prática os assuntos estudados e perceber a importância da Programação Concorrente. Por meio do uso de *locks*, variáveis condicionais e semáforos é possível gerenciar o uso de memória compartilhada entre processos, e com isso, desenvolver um programa concorrente sem condições de corridas, garantindo regiões de exclusão mútua entre as *threads*.

De forma geral, foi possível compreender durante o desenvolvimento do projeto as vantagens e os desafios que a Programação Concorrente proporcionou. A execução de processos em paralelo, muitas vezes, viabiliza o aumento do desempenho de programas, já que é possível aumentar a quantidade de tarefas executadas no mesmo tempo lógico, superando, dessa forma, as limitações da computação sequencial.

Referências

[1] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. The Benjamin/Cummings Publishing Company, Inc, 1991.

[2] M. Ben Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, 2nd edition, 2006.

[3] Andrew S. Tanenbaum. *Sistemas operacionais modernos*. Pearson Education do Brasil Ltda, 4th edition, 2016.